Using Analysis Patterns to Uncover Specification Errors

William Heaven Alessandra Russo
Department of Computing, Imperial College London
{william.heaven, ar3}@imperial.ac.uk

ABSTRACT

Developing or maintaining a formal software specification is a task unfortunately prone to the accidental introduction of logical errors, particularly inconsistencies. At worst, such errors can be dangerously misleading. For example, many software analysis tools that require a formal specification as input produce false positives when faced with inconsistency, making it more likely that developers miss errors in the software. At the same time, most existing analysis tools supporting specification development are not well suited to the detection of inconsistencies without explicit direction from an expert user. To address this shortcoming, this paper presents novel analysis "patterns" that can automatically guide specifiers through logical pitfalls of this kind by not only checking a given specification formula, but recursively checking the subformulae of that formula. By doing so, rather than present a specifier with potentially misleading feedback, use of these patterns can automatically ensure—without expert direction—that accidentally introduced inconsistencies are uncovered.

1. INTRODUCTION

Formal software specification largely remains the exclusive and sparsely-populated province of experts and enthusiasts due, in part, to the demands placed on the practitioner. Formal (typically declarative) specifications are prone to the accidental introduction of logical errors, particularly inconsistencies, during their development. Further, it is rarely the case that developing a specification is a task done once, checked, and forgotten. In practice, as software evolves, a specification must be extended with the addition of new constituent formulae, making the introduction of logical error an ongoing risk.

The difficulty is compounded by the fact that most specifications of non-trivial software systems typically contain many logical interdependencies and, therefore, the consistency of one part of a specification is likely to be affected by changes to other parts. Currently popular specification languages for component-based software (such as JML [15] and Spec♯ [1]) allow side-effect-free method calls to be used as terms in specification formula. So, for example, a side-effect-free method *size()* that returns an integer result could be used in a specification formula such as *size() < MAX*. However, while this specification-language feature has the great advantage of affording a succinct and modular specification style, it exponentially increases the number of dependencies between formulae in a specification. For example, the consistency of any formula containing the term *size()* will depend on the formulae in the method specification for *size()*, which may in turn be expressed using side-effect-free method calls and thus depend on the method specifications of those methods, and so on. Interdependencies of this sort, where the specification for a method like *size()* may not be immediately visible from the contexts in which the term *size()* is used in a formula, make it even harder to avoid introducing—and to notice—logical errors.

There are many tools available for specification analysis ranging from lightweight static- and runtime-checking tools to those that offer the potential for more heavyweight verification [1, 3, 16]. However, most concentrate on analysis of the relation between specification and code and not on analysis of the specification itself. Tools such as the SAT-based Alloy Analyzer [12] permit versatile analyses of specifications but even the Alloy Analyzer provides misleading feedback when analysing an inconsistent specification unless expertly directed. This is because the results of a consistency check will be positive (suggesting no inconsistency) when the set of formulae in question are not only consistent but vacuously consistent—"valid"—on account of inconsistent subformulae. For example, a formula $\phi \Rightarrow \psi$ is valid if ϕ is inconsistent. In this case, the logical error causing ϕ's inconsistency is "hidden" by the positive result of the consistency check.

What is needed are powerful automated tools to support the developers and maintainers of a specification. As a step towards this goal, this paper presents a set of analysis "patterns" that guide specifiers through the pitfalls of logical analysis by not only checking the consistency of a given specification formula but recursively checking the subformulae of that formula. By doing so, rather than present a specifier with potentially misleading feedback, use of these patterns can automatically ensure—without expert direction—that accidentally introduced inconsistencies are uncovered. An implementation of these patterns, using the Alloy Analyzer as a backend, has also been developed [10].

The analysis patterns are presented in the context of satisfiability-based analysis and, following a motivating example in Section 2, some preliminaries to their presentation are set forth in Sections 3 and 4. Section 5 then presents the patterns themselves. An implementation is briefly discussed in Section 6 before briefly considering some related work in Section 7. Finally, Section 8 concludes.

2. MOTIVATING EXAMPLE

One problem in evolving software is that it is often possible to adversely affect existing code by adding something new. Where that new thing is a subtype, ensuring that the subtype is a behavioural subtype [17] is a good way to avoid introducing undesirable behaviour. Behavioural subtyping effectively guarantees that the addition of a subtype does not affect the behaviour of the existing program. A behavioural subtype can be substituted for its supertype without observable difference in program behaviour. In specification languages such as JML and Spec♯, behavioural subtyping can be enforced via specification inheritance whereby the specification of a subtype implicitly includes that of its supertype [5].

Consider a Java class *Queue* with a boolean method *insert()*. If *entries* is the data structure in *Queue* representing the queued elements, a postcondition for *insert()* might be specified in JML as follows (the JML specifications here are expressed using boolean Java expressions plus the standard propositional operators and the $\backslash$*old* keyword denoting pre-state values; they appear between special "/*@ ... @*/" comments):

```
/*@ ensures (result ==> contains(e))
        && (entries == \old(entries.add(e))); @*/
boolean insert(Entry e) { ... }
```

This says two things. Firstly, that the boolean result of inserting element *e* implies the boolean result of a call to *contains()* on the same *Queue* object. Secondly, that *entries* after a call to *insert()* is equal to *entries* before the call in all respects other than *e*'s addition. In other words, the only difference between post- and pre-state *entries* is that *e* is added: all other elements in *entries* remain the same.

Assume that an evolution of the software containing *Queue* involves adding a subtype *BoundedQueue* which adds an extra method *size()* and overrides *insert()*. These additions might be specified as follows:

```
/*@ ensures size() == entries.size(); @*/
/*@ pure @*/ int size() { ... }

/*@ also
    ensures size() < \old(entries.size()) && size() <= MAX; @*/
boolean insert(Entry e) { ... }
```

The specification for *size()* says that its integer result will always equal the result of a call to the *size()* method of *entries*. Note also that *size()* is specified to be side-effect free with the keyword *pure*. This means that *size()* can be used as a term in the postcondition of the overriding *insert()*. The *also* keyword highlights that this new postcondition is considered in conjunction with the postcondition of the overridden *insert()*. Thus, the postcondition of *insert()* in *BoundedQueue* is the postcondition of the overridden method plus the above, which says that the size of the queue after a call to *insert()* is less than the pre-state value of *entries.size()* and less than or equal to some given value *MAX* (it is assumed, without going into detail, that when the queue is already full the insertion does not take place and the size of the queue does not change).

For *BoundedQueue* to be a behavioural subtype of *Queue*, the postcondition of *insert()* in *BoundedQueue* must imply the postcondition of *insert()* in *Queue*. This should be enforced by the fact that the overriding *insert()* includes the postcondition of the overridden *insert()* and the implication can be checked in a tool such as the Alloy Analyzer. As might be expected, the result given by the Alloy Analyzer in this case is that the implication is valid.

However, this positive result is misleading because it hides a logical error in the specification of *insert()* in *BoundedQueue*. This postcondition says that following a call to *insert()*, the size of the queue is *less than* the pre-state value of *entries.size()*, which is inconsistent with the specification of *size()* and the existing *insert()* postcondition. Following a call to *insert()* the size of the queue cannot be less than the pre-state value of *entries.size()*.

What is needed to uncover hidden specification errors in cases such as this is a means to analyse not only the top-level formula but also its subformulae. Specification errors are often not detected explicitly through a consistency check of top-level formulae alone. In this case, an automated analysis should detect not only that the implication is valid (vacuously consistent) but further investigate the subformulae of the analysed formula—the constituent formulae not only of the *insert()* method specification but also of the *size()* method specification—to uncover the source of the validity, which for an implication is possibly an inconsistent antecedent.

3. SATISFIABILITY VALUES

Establishing the consistency of a specification formula can be considered an instance of the Boolean Satisfiability Problem (SAT) [18, 8]. A formula ϕ in a specification language with a well-defined semantics is said to be *satisfiable* iff there is a possible assignment of values in that semantics to the terms of ϕ (variables, constants, and side-effect-free method calls) that makes ϕ true. Conversely, ϕ is said to be *unsatisfiable* iff there is no such assignment, i.e., for every possible assignment ϕ is false. Henceforth, **s** will denote the value *satisfiable* and **u** will denote the value *unsatisfiable*.

Satisfiability analysis in practice has well known limitations and automatic decision procedures for deciding satisfiability tend to be incomplete. While these limitations will be touched on briefly in Section 6, an ideal satisfiability procedure, or "oracle", will be assumed for clear and succinct presentation of the analysis patterns. This oracle is deemed to be sound and complete.

Definition 1 (Satisfiability Oracle). Let Φ denote the set of formulae in a specification language. An ideal satisfiability procedure, or *oracle*, is represented by the function $SAT : \Phi \longrightarrow \{\mathbf{s}, \mathbf{u}\}$ such that, for a given formula $\phi \in \Phi$, $SAT\ (\phi) = \mathbf{s}$ iff ϕ is satisfiable and $SAT\ (\phi) = \mathbf{u}$ otherwise.

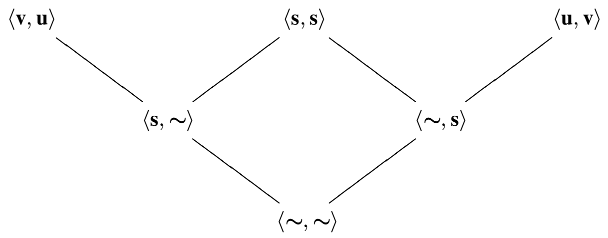
For a formula ϕ, $\mathbf{s}\ (\phi)$ will denote that $SAT\ (\phi) = \mathbf{s}$ and $\mathbf{u}\ (\phi)$ will denote that $SAT\ (\phi) = \mathbf{u}$.

The single query $SAT\ (\phi)$ is sufficient to decide whether $\mathbf{s}\ (\phi)$ or $\mathbf{u}\ (\phi)$. Further, if a formula ϕ is unsatisfiable, i.e., false for all assignments, then $\neg\phi$ must be true for all assignments. A formula that is true for all assignments is said to be *valid*. Therefore, to discover that ϕ is unsatisfiable is also to discover that $\neg\phi$ is valid. On the other hand, if a formula is only true for some assignments but not all, i.e., there are some assignments for which its negation is true, then the formula is said to be *contingent*. Thus, discovering that ϕ and $\neg\phi$ are both satisfiable is to discover that both are contingent. Finally, if a formula ϕ is known to be satisfiable, $\neg\phi$ cannot be valid because this would contradictorily require ϕ to be unsatisfiable. Therefore, a value of *not valid* can be established, dual to satisfiable (*satisfiable* is of course equivalent to *not unsatisfiable*). The values *valid*, *contingent*, and *not valid* can be defined in terms of *SAT*.

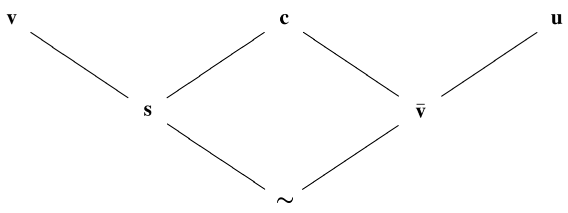
Definition 2 (Valid, Contingent, and Not Valid). Given a formula ϕ, ϕ is *valid* iff $\mathbf{u}\ (\neg\phi)$; ϕ is *contingent* iff both $\mathbf{s}\ (\phi)$ and $\mathbf{s}\ (\neg\phi)$; and ϕ is *not valid* if $\mathbf{s}\ (\neg\phi)$.

The values *valid*, *contingent*, and *not valid* will be denoted by **v**, **c**, and $\bar{\mathbf{v}}$, respectively. Further, for a formula ϕ, $\mathbf{v}\ (\phi)$ will denote that ϕ is known to be valid, $\mathbf{c}\ (\phi)$ will denote that ϕ is known to be contingent, and $\bar{\mathbf{v}}\ (\phi)$ will denote that ϕ is known to be not valid.

If knowing the satisfiability of both ϕ and $\neg\phi$ is to have full information regarding the satisfiability of ϕ (and, symmetrically, regarding the satisfiability of $\neg\phi$) and knowing the satisfiability of ϕ but not $\neg\phi$ (or, conversely, $\neg\phi$ but not ϕ) is to have partial information regarding the satisfiability of ϕ, then it can be said that knowing neither the satisfiability of ϕ nor $\neg\phi$ is to have no information regarding the satisfiability of ϕ. If nothing is known about the



(a) Pairs of formula and negation



(b) Satisfiability values

Figure 1: Orderings for (a) pairs of formula and negation and (b) satisfiability values

satisfiability of ϕ or $\neg\phi$ the value of both formulae can be given the value *not known*. The value *not known* will be denoted by $\sim$ and, for a formula ϕ, $\sim(\phi)$ will denote that no value for ϕ or $\neg\phi$ is yet known. Summing up, the set of *satisfiability values* can be defined.

Definition 3 (Satisfiability Values). The set of *satisfiability values* is the set *SatVal* = $\{\mathbf{v}, \mathbf{c}, \mathbf{u}, \mathbf{s}, \bar{\mathbf{v}}, \sim\}$. The *satisfiability variable* ν ranges over *SatVal*. A *satisfiability claim* for a formula ϕ is an expression $\nu\ (\phi)$ in which ν is instantiated with one of the values in *SatVal*.

For example, the satisfiability claim $\mathbf{v}\ (\phi)$ is true iff ϕ is known to be valid and the satisfiability claim $\sim(\phi)$ is true iff no satisfiability value for ϕ is known.

The three possibilities with respect to knowing the satisfiability of a formula and its negation, viz., full information, partial information, and no information, provide the basis for a partial ordering of satisfiability values according to what might be called *information content*. The partial ordering of satisfiability value pairs for a formula ϕ and its negation is shown in Figure 1 (a). The more information contained in a pair of values, the higher the pair is in the ordering. For instance, the pair $\langle\sim,\sim\rangle$ represents having no information about the satisfiability of either ϕ or $\neg\phi$, the pair $\langle\sim,\mathbf{s}\rangle$ represents the information that $\neg\phi$ is satisfiable, and the pair $\langle\mathbf{s},\mathbf{s}\rangle$ represents the information that both ϕ or $\neg\phi$ are satisfiable.

Certain possible pairings are obviously omitted, some because they are redundant. The four pairs $\langle\sim,\mathbf{v}\rangle$, $\langle\sim,\mathbf{u}\rangle$, $\langle\mathbf{u},\sim\rangle$ and $\langle\mathbf{v},\sim\rangle$ are omitted because in each case the value $\sim$ can trivially be replaced by $\mathbf{v}$ or $\mathbf{u}$ according to the value of the other element in the pair. For example, the $\sim$ in $\langle\sim,\mathbf{v}\rangle$ can immediately be replaced by $\mathbf{u}$ since if $\neg\phi$ is true for all assignments then ϕ is true for none. Similarly, the two pairs $\langle\mathbf{u},\mathbf{s}\rangle$ and $\langle\mathbf{s},\mathbf{u}\rangle$ are omitted because in each case the value $\mathbf{s}$ can be replaced by $\mathbf{v}$. Finally, the four pairs $\langle\mathbf{v},\mathbf{v}\rangle$, $\langle\mathbf{u},\mathbf{u}\rangle$, $\langle\mathbf{s},\mathbf{v}\rangle$ and $\langle\mathbf{v},\mathbf{s}\rangle$ represent impossible situations. For example, a formula cannot be valid if its negation is satisfiable.

The ordering of Figure 1 (a) can also be represented with respect to the values of *SatVal*, as in Figure 1 (b). A partial ordering is thus defined for *SatVal* .

Definition 4 (Ordering of Satisfiability Values). The set *SatVal* is partially ordered according to information content as follows:

$$\mathbf{v} > \mathbf{s}, \quad \mathbf{c} > \mathbf{s}, \quad \mathbf{c} > \bar{\mathbf{v}}, \quad \mathbf{u} > \bar{\mathbf{v}}, \quad \mathbf{s} > \sim, \quad \bar{\mathbf{v}} > \sim$$

For all values $\nu_1, \nu_2 \in$ *SatVal*, ν_1 is said to be *more precise* (resp. *less precise*) if and only if $\nu_1 > \nu_2$ (resp. $\nu_2 > \nu_1$).

The ordering is assumed to have the usual concept of *least upper bound*, i.e., for two values $\nu_1, \nu_2 \in$ *SatVal*, the least upper bound of ν_1 and ν_2, written $\nu_1 \sqcup \nu_2$, if defined, is the unique value $\nu_3 \in$ *SatVal* such that $\nu_3 \geq \nu_1$ and $\nu_3 \geq \nu_2$ and for all other values $\nu_4 \in$ *SatVal*, if $\nu_4 \geq \nu_1$ and $\nu_4 \geq \nu_2$, then $\nu_4 \geq \nu_3$. For example, $\mathbf{s} \sqcup \sim = \mathbf{s}$, $\mathbf{s} \sqcup \bar{\mathbf{v}} = \mathbf{c}$ and $\mathbf{v} \sqcup \mathbf{v} = \mathbf{v}$, but $\mathbf{v} \sqcup \mathbf{c}$, $\mathbf{v} \sqcup \mathbf{u}$, and $\mathbf{c} \sqcup \mathbf{u}$ do not exist.

4. OBTAINING SATISFIABILITY VALUES

It was noted that the single query *SAT* (ϕ) decides only whether $\mathbf{s}\ (\phi)$ or $\mathbf{u}\ (\phi)$. However, in certain cases satisfiability values other than $\mathbf{s}$ and $\mathbf{u}$ can be obtained through inference. For example, if the result $\mathbf{s}\ (\phi)$ has been previously established, then a new result $\mathbf{s}\ (\neg\phi)$ would allow both $\mathbf{c}\ (\phi)$ and $\mathbf{c}\ (\neg\phi)$ to be inferred. Or, if the new result is $\mathbf{u}\ (\neg\phi)$, then $\mathbf{v}\ (\phi)$ can be inferred. A full algebra defining the possible inferences for the values in *SatVal* is documented in [10]. The patterns of analysis described in the next section make use of a *lookup table* that records satisfiability values for formulae as they are discovered during analysis. The lookup table is a map from formulae to satisfiability values, obtained either by satisfiability queries or inferred from previous results. Initially, all formulae are mapped to the value $\sim$.

Definition 5 (Satisfiability Lookup Table). A *satisfiability lookup table* is a map $SAT_{Table} : \Phi \rightarrow SatVal$ from formulae to satisfiability values.

The value of a formula ϕ can now be obtained by querying either the satisfiability oracle or the lookup table. Either way, it is desirable to obtain the more precise value. For example, when querying the satisfiability value of ϕ, if $SAT\ (\phi) = \mathbf{s}$ but $SAT_{Table}\ (\phi) = \mathbf{v}$ (which would be the case when the more precise value $\mathbf{v}$ had been previously inferred for ϕ), then the value $\mathbf{v}$ should be taken, since $\mathbf{v} > \mathbf{s}$. Given two satisfiability values, ν_1 and ν_2, the most precise value obtainable is the least upper bound of the two, i.e., $\nu_1 \sqcup \nu_2$. Note that the most precise value need not in fact be either ν_1 or ν_2. For example, if $\nu_1 = \mathbf{s}$ and $\nu_2 = \bar{\mathbf{v}}$, then the least upper bound, and therefore most precise value obtainable, is $\mathbf{c}$. A lookup table is sound in the sense that the least upper bound always exists for a given update. The following function gives the value of a formula.

Definition 6(Obtaining the Satisfiability Value of a Formula). Given an oracle *SAT* and satisfiability lookup table SAT_{Table}, the function

$$GetVal : \Phi \rightarrow SatVal$$

gives a satisfiability value for a formula, such that, for all $\phi \in \Phi$

$$GetVal\ (\phi) = SAT\ (\phi) \sqcup SAT_{Table}\ (\phi).$$

However, as new values are learnt for a formula ϕ, a lookup table may need to be updated so that the value obtained through *GetVal* is always the most precise value yet discovered in a given analysis. Occasionally, a value may be inferred for a formula during analysis that is less precise than that already recorded in the lookup table. For example, if a conjunction $\phi_1 \wedge \phi_2$ is found to be satisfiable, it is implied that the conjuncts ϕ_1 and ϕ_2 are also satisfiable. However, if lookup table already maps ϕ_1 to **v**, then this entry should not be updated with the value **s**. The lookup table mapping should never be updated with a value for ϕ that is less precise than its existing value. An appropriate update function is defined below.

Definition 7(Lookup Table Update). For a formula ϕ, satisfiability value $\nu \in \mathit{SatVal}$, and lookup table $\mathit{SAT}_{\mathit{Table}}$, an *updated lookup table* $\mathit{SAT}'_{\mathit{Table}}$ is given by the function $\mathit{Upd}\ (\mathit{SAT}_{\mathit{Table}}, \phi, \nu)$ such that

$$\mathit{Upd}\ (\mathit{SAT}_{\mathit{Table}}, \phi, \nu) = \mathit{SAT}_{\mathit{Table}} \oplus \phi \mapsto \nu$$

iff $\nu > \mathit{SAT}_{\mathit{Table}}\ (\phi)$. Otherwise $\mathit{Upd}\ (\mathit{SAT}_{\mathit{Table}}, \phi, \nu) = \mathit{SAT}_{\mathit{Table}}$.

5. PATTERNS

Application of the analysis patterns starts with the analysis pattern for the top-level formula being queried. As analysis moves to the subformulae of the formula in question the patterns are applied recursively according to the top-level connective of whatever subformula is currently being analysed. Each satisfiability query for a formula ϕ is represented by *?* (ϕ). Each satisfiability query *?* (ϕ) is resolved by a corresponding call to *GetVal* (ϕ), the result of which determines the next formula to be analysed as dictated by the patterns. The progression of satisfiability queries from formula to formula is not necessarily a linear sequence. It is often the case that analysis of a formula branches into parallel analyses of its subformulae.

There is an analysis pattern for each of the propositional connectives common to most popular specification languages (e.g. JML, Spec♯, Alloy): *Negation Pattern*, *Conjunction Pattern*, *Disjunction Pattern*, and *Implication Pattern*. While it may be possible to consider only a basic set of patterns in which implication and either conjunction or disjunction are reducible to the remaining connectives, the full set is presented here for clarity. Analysis of an implication in particular is less straightforwardly presented without its corresponding pattern.

Analysis of a given formula is guided step by step by repeated application of the patterns, beginning with the pattern that matches the root connective of that formula. For example, the pattern matching formulae of the form $\phi_1 \Rightarrow \phi_2$ dictates that ϕ_1 and $\neg\phi_2$ should be checked for satisfiability if $\phi_1 \Rightarrow \phi_2$ is valid. Where application of a pattern identifies that a formula is valid or unsatisfiable due to a valid or unsatisfiable subformula, analysis terminates with a warning accompanied by a reference to the subformulae in which a potential error may reside. There is also a pattern for atomic (connective-free) formulae known as the *Base Pattern*. Application of the Base Pattern to a contingent atomic formula results in default (i.e., *warning-free*) termination of a decompositional analysis process.

The patterns are represented as the decision diagrams shown in Figures 3–7. In these diagrams, a non-terminal node represents a satisfiability query, e.g., *?* (ϕ), and, for two nodes *A* and *B*, a transition from *A* to *B* represents an application of *GetVal* (ϕ), where ϕ is the formula contained in the satisfiability query of *A*. Transitions are labelled with a satisfiability value in *SatVal*. A transition labelled with satisfiability value ν is taken from a node representing the satisfiability query *?* (ϕ) iff *GetVal* $(\phi) = \nu$. For example, a transition labelled **v** is taken from a node representing the satisfiability query *?* (ϕ) iff *GetVal* $(\phi) = \mathbf{v}$. Where there are multiple transitions from node *A* to node *B*, a single transition is shown but with multiple labels.

The notation used in the diagrams of Figures 4–7 is summarised in Figure 2. Non-terminal nodes are depicted as shown in Figure 2 (a). The start node of each pattern is known as the *pattern root* and depicted as shown in Figure 2 (b). A pattern root repre-

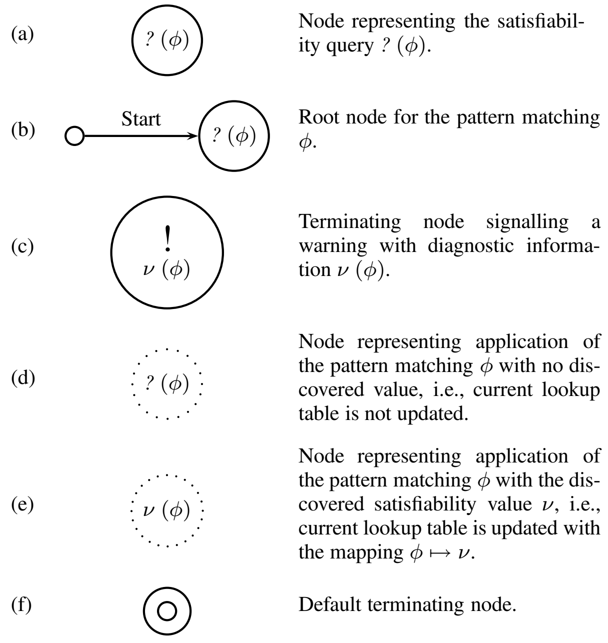


Figure 2: Summary of Pattern Notation

sents a satisfiability claim for a formula whose top-level connective is matched by the pattern. For instance, the root of the Implication Pattern (Figure 7) contains the satisfiability claim *?* $(\phi_1 \Rightarrow \phi_2)$. A pattern is said to be *applied* to the formula in its root and the formula in the root is known as the *root formula* of the pattern.

A terminal node in a pattern is known as a *pattern leaf*. There are three kinds of pattern leaf:

- Leaf signalling a warning
- Leaf representing an application of a further pattern
- End leaf

A leaf that signals a warning is depicted as shown in Figure 2 (c). If application of a pattern terminates with a warning, it signals a potential specification error. A warning is issued in the following cases:

- An atomic formula is valid
- An atomic formula is unsatisfiable
- An implication is valid due to an unsatisfiable antecedent
- An implication is valid due to a valid consequent

For example, if an atomic formula contains a pure method term it will be unsatisfiable if the pure method term is undefined for all program assignments. Further, an atomic formula can be valid if it contains a pure method term whose precondition and postcondition are valid. A warning node also contains diagnostic information in the form of a satisfiability claim. For example, in the application of an Implication Pattern to a formula $\phi_1 \Rightarrow \phi_2$, a warning may be issued with the satisfiability claim $\mathbf{u}\ (\phi_1)$ indicating that the root formula is possibly valid due to an error in the specification of ϕ.

A leaf that represents an application of a further pattern is depicted as shown in Figure 2 (d)-(e). Such a termination of a pattern means that the pattern for the formula ϕ should be applied to investigate further. The node may contain a satisfiability query, as in Figure 2 (d), indicating that the satisfiability value of ϕ may not be known. However, in some cases, application of the pattern leading

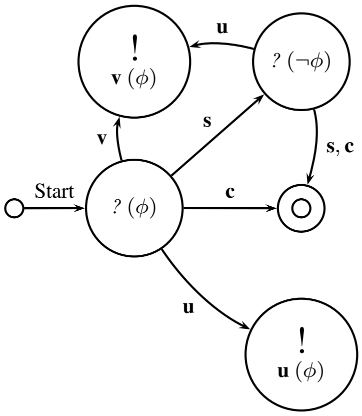


Figure 3: Base Pattern

to this leaf node may have provided a value for ϕ. In this case, the leaf node may contain a satisfiability claim, e.g., $\mathbf{v}$ (ϕ), as in Figure 2 (e). This indicates that the given satisfiability value, e.g., $\mathbf{v}$, is entered into the satisfiability lookup table for ϕ, i.e., a new mapping $SAT_{Table}{}' = SAT_{Table} \oplus \phi \mapsto \mathbf{v}$ is constructed. In this way, information about the satisfiability of formula is accumulated through recursive application of the patterns. Finally, an end leaf is depicted as shown in Figure 2 (f). An end leaf appears only once, in the representation of the Base Pattern, and represents default termination of an application of the patterns.

Base Pattern. The Base Pattern (Figure 3) is applied to atomic formulae. If an atomic formula ϕ is unsatisfiable (i.e., $\mathit{GetVal}\ (\phi) = \mathbf{u}$), the transition labelled $\mathbf{u}$ is taken from the pattern root and a warning is issued signalling the unsatisfiability of ϕ. Similarly, if ϕ is known to be valid (i.e., $\mathit{GetVal}\ (\phi) = \mathbf{v}$), the transition labelled $\mathbf{v}$ is taken and a warning is issued signalling the validity of ϕ. If ϕ is known to be contingent (i.e., $\mathit{GetVal}\ (t) = \mathbf{c}$), the transition labelled $\mathbf{c}$ is taken and application of the pattern ends normally since a term being true in some states and false in others is as expected. Otherwise, ϕ is only known to be satisfiable (i.e., $\mathit{GetVal}\ (t) = \mathbf{s}$), and the transition labelled $\mathbf{s}$ is taken. In this case, the negation of ϕ is then checked for satisfiability to determine whether or not ϕ is valid. If $\neg\phi$ is satisfiable (or known to be contingent, as indicated by the label $\mathbf{s}$, $\mathbf{c}$), then ϕ is contingent and no warning need be issued: application of the pattern terminates normally. But if $\neg\phi$ is unsatisfiable, then ϕ is valid and, as above, a warning signalling the validity of ϕ is again issued. Note that the check for the satisfiability of $\neg\phi$ is included in the Base Pattern and not treated as an application of the Negation Pattern. This is to avoid cycling between the Base Pattern, which given ϕ checks $\neg\phi$, and the Negation Pattern, which given $\neg\phi$ checks ϕ.

Negation Pattern. The Negation Pattern (Figure 4) is applied to formulae of the form $\neg\phi$. Application of this pattern permits the exploration of whether or not $\neg\phi$ is valid or unsatisfiable and hence, conversely, whether or not ϕ is unsatisfiable or valid. If a formula $\neg\phi$ is unsatisfiable (i.e., $\mathit{GetVal}\ (\neg\phi) = \mathbf{u}$), the transition labelled $\mathbf{u}$ is taken from the pattern root. Since ϕ has been discovered to be valid, the pattern matching the formula ϕ is applied to investigate further. If $\neg\phi$ is known to be valid (i.e., $\mathit{GetVal}\ (\neg\phi) = \mathbf{v}$), the transition labelled $\mathbf{v}$ is taken and since ϕ is discovered to be unsatisfiable, the pattern matching the formula ϕ is applied to investigate further. If $\neg\phi$ is known to be contingent (i.e., $\mathit{GetVal}\ (\neg\phi) = \mathbf{c}$), then ϕ must be contingent. Though a formula ϕ is expected to be contingent, valid and unsatisfiable subformulae can still hide

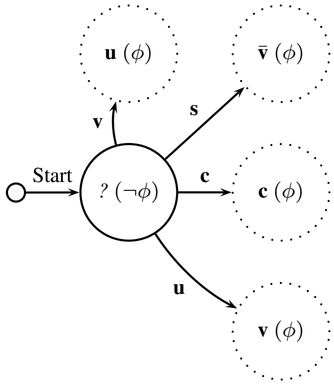


Figure 4: Negation Pattern

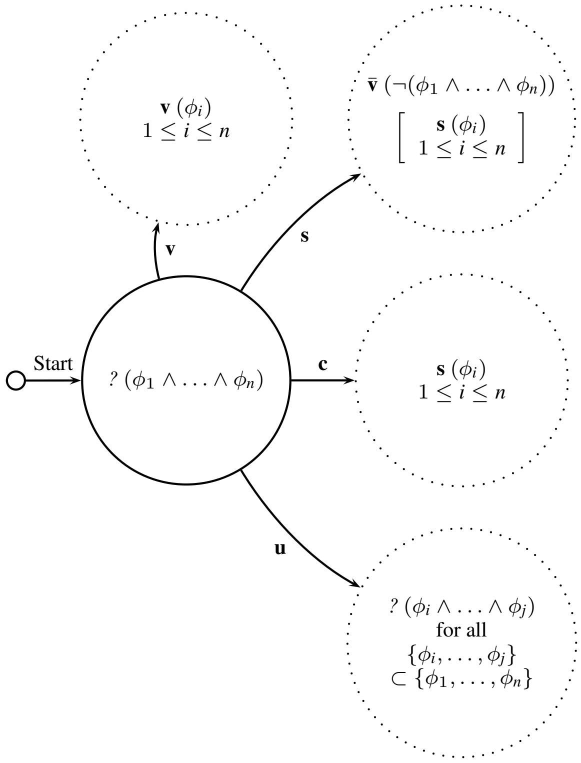


Figure 5: Conjunction Pattern

beneath a contingent formula, so ϕ may be investigated further through application of its pattern. Otherwise, $\neg\phi$ is known only to be satisfiable (i.e., $\mathit{GetVal}\ (\neg\phi) = \mathbf{s}$), implying that ϕ cannot be valid (though possibly unsatisfiable), i.e., $\bar{\mathbf{v}}$ (ϕ). ϕ can again be further investigated through application of its pattern. Note that the Negation Pattern does not check the negation of its root formula since analysing $\neg\neg\phi$ is of course equivalent to analysing ϕ, which would be redundant given that ϕ is always next analysed through application of its pattern.

Conjunction Pattern. The Conjunction Pattern (Figure 5) is applied to formulae of the form $\phi_1 \wedge \ldots \wedge \phi_n$. Application of this pattern decomposes a conjunction into its conjuncts to identify whether the conjunction is valid or, if it is unsatisfiable, which conjuncts or combination of conjuncts are unsatisfiable. If $\phi_1 \wedge \ldots \wedge \phi_n$

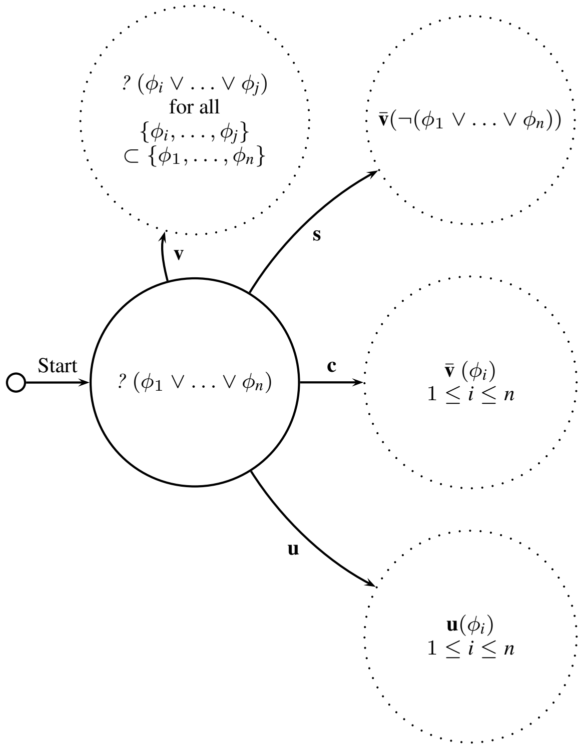


Figure 6: Disjunction Pattern

is unsatisfiable (i.e., *GetVal* $(\phi_1 \wedge \ldots \wedge \phi_n) = \mathbf{u}$), the transition labelled **u** is taken from the pattern root. However, it is not necessarily the case that any single conjunct is unsatisfiable. Investigating further by simply applying the pattern for each conjunct is insufficient since a conjunction can be unsatisfiable even though each conjunct is satisfiable in isolation. This is because two conjuncts ϕ_i and ϕ_j, say, may be inconsistent. In other words, ϕ_i may imply $\neg\phi_j$ and vice versa. To investigate an unsatisfiable conjunction further, the patterns for all arbitrary subsets of conjunct are applied. In this case, no value for each combination is yet implied.

On the other hand, if $\phi_1 \wedge \ldots \wedge \phi_n$ is known to be valid (i.e., *GetVal* $(\phi_1 \wedge \ldots \wedge \phi_n) = \mathbf{v}$), the transition labelled **v** is taken. This transition implies that each conjunct ϕ_i is valid and the validity of ϕ_i, for $1 \leq i \leq n$ is investigated through application of its pattern. If $\phi_1 \wedge \ldots \wedge \phi_n$ is known to be contingent (i.e., *GetVal* $(\phi_1 \wedge \ldots \wedge \phi_n) = \mathbf{c}$), the transition labelled **c** is taken. For the conjunction to be contingent, no conjunct can be unsatisfiable, i.e., either $\mathbf{v}\ (\phi)$ or $\mathbf{c}\ (\phi)$ for each ϕ_i. Moreover, since it is known that the conjunction is not valid, at least one conjunct (and possibly all) must be contingent. However, which conjuncts are contingent and which valid, if any, cannot be determined without further application of the patterns for each ϕ_i. The most that can be stated at this point about each ϕ_i is that it is not unsatisfiable, i.e., $\mathbf{s}\ (\phi_i)$.

Finally, if $\phi_1 \wedge \ldots \wedge \phi_n$ is known only to be satisfiable (i.e., *GetVal* $(\phi_1 \wedge \ldots \wedge \phi_n) = \mathbf{s}$), the pattern for its negation is applied to determine whether or not $\phi_1 \wedge \ldots \wedge \phi_n$ is valid. The lookup table can here be augmented in two ways. First, with the value $\bar{\mathbf{v}}$ (not valid) for $\neg(\phi_1 \wedge \ldots \wedge \phi_n)$ since the root formula is known to be satisfiable, and second, with the value **s** for each ϕ_i since, if the conjunction is satisfiable so are each of its conjuncts. In Figure 5, this second update is shown in parenthesis.

Disjunction Pattern. The Disjunction Pattern (Figure 6) is applied to formulae of the form $\phi_1 \vee \ldots \vee \phi_n$. Application of this pattern decomposes a disjunction into its disjuncts to identify whether the disjunction is valid or unsatisfiable, and if valid, which disjuncts or combination of disjuncts are valid. The Disjunction Pattern is the dual of the Conjunction Pattern in that the valid (resp. unsatisfiable) case of the Disjunction Pattern is the dual of the unsatisfiable (resp. valid) case of the Conjunction Pattern. Considering each case in turn, if $\phi_1 \vee \ldots \vee \phi_n$ is *unsatisfiable* (i.e., *GetVal* $(\phi_1 \vee \ldots \vee \phi_n) = \mathbf{u}$) the transition labelled **u** is taken from the pattern root. For the disjunction to be unsatisfiable all disjuncts must be unsatisfiable and the pattern for each ϕ_i, $1 \leq i \leq n$ is applied to investigate further. The lookup table is updated to map each ϕ_i to the value **u**.

If $\phi_1 \vee \ldots \vee \phi_n$ is known to be *valid* (i.e., *GetVal* $(\phi_1 \vee \ldots \vee \phi_n) = \mathbf{v}$) the transition labelled **v** is taken. Note that $\phi_1 \vee \ldots \vee \phi_n$ may be valid not only on account of a valid disjunct, but because two disjuncts ϕ_i and ϕ_j, say, may be inconsistent such that ϕ_i implies $\neg\phi_j$ and vice versa. Therefore, to investigate a valid disjunction further, the patterns for all arbitrary subsets of disjunct are applied. No value for each combination is implied so the lookup table is not updated for any $\phi_i \vee \ldots \vee \phi_j$.

If $\phi_1 \vee \ldots \vee \phi_n$ is known to be *contingent* (i.e., *GetVal* $(\phi_1 \vee \ldots \vee \phi_n) = \mathbf{c}$), the transition labelled **c** is taken. Since the disjunction is not valid, no disjunct can be valid (i.e., $\bar{\mathbf{v}}\ (\phi_i)$, for $1 \leq i \leq j \leq n$) and the lookup table is updated with the value $\bar{\mathbf{v}}$ for each ϕ_i before applying the pattern for each ϕ_i to investigate further. Finally, if it is known only that $\phi_1 \vee \ldots \vee \phi_n$ is *satisfiable* (i.e., *GetVal* $(\phi_1 \vee \ldots \vee \phi_n) = \mathbf{s}$) and the transition labelled **s** is taken. To determine whether or not the root formula is valid the pattern for $\neg(\phi_1 \vee \ldots \vee \phi_n)$ is applied. Again, in this case, it is known that no disjunct can be valid and the lookup table is updated to map each ϕ_i to $\bar{\mathbf{v}}$. Note that this case is the dual of the unsatisfiable case in the Conjunction Pattern.

Implication Pattern. The Implication Pattern (Figure 7) is applied to formulae of the form $\phi_1 \Rightarrow \phi_2$. Application of this pattern investigates whether its root formula is valid because of an unsatisfiable antecedent or valid consequent or, if the root formula is unsatisfiable, why the antecedent is valid and consequent unsatisfiable. If $\phi_1 \Rightarrow \phi_2$ is *unsatisfiable* (i.e., *GetVal* $(\phi_1 \Rightarrow \phi_2) = \mathbf{u}$), *both* transitions labelled **u** are taken from the pattern root. For the implication to be unsatisfiable, ϕ_1 must be valid and ϕ_2 must be unsatisfiable. The validity of ϕ_1 and the unsatisfiability of ϕ_2 can be investigated further through application of the respective patterns for each.

If $\phi_1 \Rightarrow \phi_2$ is known to be *valid* (i.e., *GetVal* $(\phi_1 \Rightarrow \phi_2) = \mathbf{v}$), both transitions labelled **v** are taken. The validity of the root formula must be due either to the validity of ϕ_1 or the unsatisfiability of ϕ_2 (or both). On one branch, the satisfiability of ϕ_1 is checked. If ϕ_1 is unsatisfiable, a warning is issued indicating that $\phi_1 \Rightarrow \phi_2$ is valid on account of an unsatisfiable antecedent. Otherwise, if ϕ_1 is found to be any of valid, contingent or satisfiable (i.e., *GetVal* $(\phi_1) \in \{\mathbf{s}, \mathbf{v}, \mathbf{c}\}$), the formula can be investigated further through application of its pattern. If new values for ϕ_2 and $\neg\phi$ are known, the lookup table is updated. On the second branch labelled **v**, the satisfiability of $\neg\phi_2$ is checked. If $\neg\phi_2$ is unsatisfiable, a warning is issued indicating that $\phi_1 \Rightarrow \phi_2$ is valid on account of a valid consequent. Otherwise, if $\neg\phi_2$ is found to be any of any of valid, contingent or satisfiable, i.e., *GetVal* $(\phi_1) \in \{\mathbf{s}, val, \mathbf{c}\}$, the formula ϕ_2 is not valid and it can be investigated further through application of its pattern. Again, if new values for ϕ_2 and $\neg\phi$ are known, the lookup table is updated.

On the other hand, if $\phi_1 \Rightarrow \phi_2$ is known to be *contingent*, i.e., *GetVal* $(\phi_1 \Rightarrow \phi_2) = \mathbf{c}$, the transition labelled **c** is taken from the pattern root. In this case, nothing is known about the values

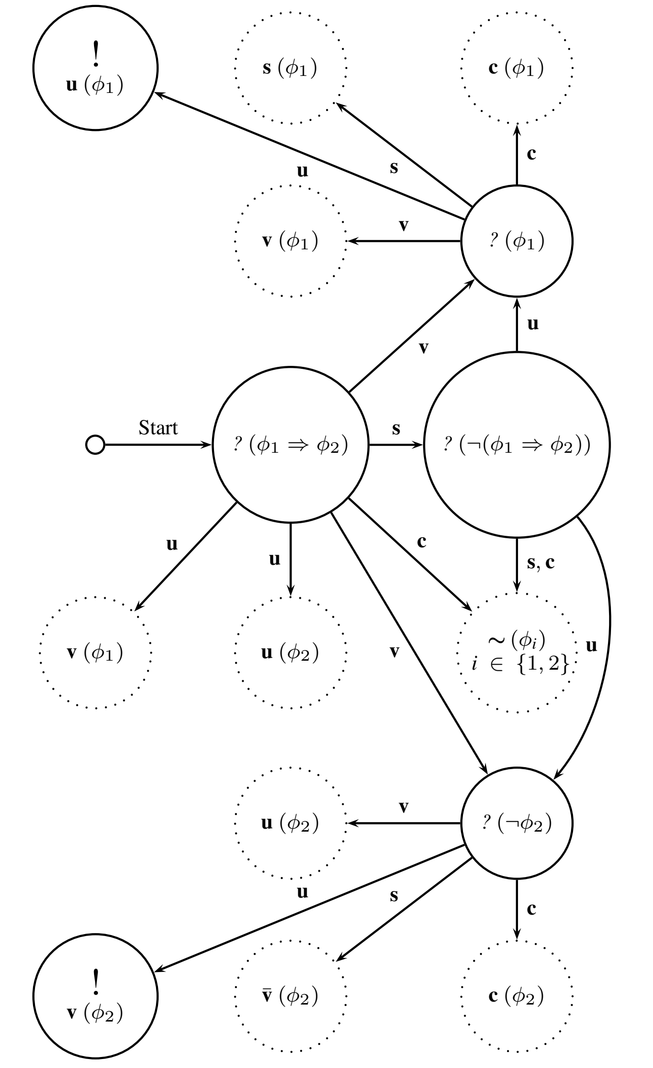


Figure 7: Implication Pattern

of ϕ_1 and ϕ_2, further investigation may be carried out through application of the patterns for these formulae. Finally, if $\phi_1 \Rightarrow \phi_2$ is known only to be *satisfiable*, i.e., *GetVal* $(\phi_1 \Rightarrow \phi_2) = \mathbf{s}$, the transition labelled **s** is taken and the satisfiability of the negation of the root formula is checked. If the negation is satisfiable, the root formula and its negation are contingent and the patterns for ϕ_1 and ϕ_2 may be applied. If the negation is unsatisfiable, then $\phi_1 \Rightarrow \phi_2$ is discovered to be valid and the steps described above are taken.

6. IMPLEMENTATION AND EVALUATION

A prototype tool has been developed to evaluate the effectiveness of the patterns in practice. The tool automatically analyses formulae of a specification language called *Loy*, which is essentially a subset of JML. Details of Loy and the tool are given in [10]. The tool harnesses the SAT-solving capabilities of the Alloy Analyzer by encoding Loy specifications into the Alloy input language and then running satisfiability queries on this encoding. The results of this analysis are then fed back to the user in terms of the original Loy specification.

Returning to the example of Section 2, it is now possible to illustrate how an automated pattern-driven analysis can uncover the hidden inconsistency introduced during the evolution of the method specification. For clarity, some of JML's ascii syntax is ignored in the following (e.g. $\Rightarrow$ is written for $==>$ and $\wedge$ for &&). The top-level formula to be analysed is the implication to test that the postcondition of *insert* in *BoundedQueue* implies that of *insert* in *Queue*. The Implication Pattern is applied to this formula and the first satisfiability query finds it to be satisfiable. The branch labelled **s** is taken from the start node and so the satisfiability of the formula's negation is queried next. The negation is found to be unsatisfiable. At this point, the Implication Patten splits and analysis of the antecedent and consequent continue in parallel. Satisfiability of the antecedent is queried and it is found to be unsatisfiable:

u *((result $\Rightarrow$ contains(e)) $\wedge$ entries = $\backslash$old(entries.add(e)*
$\wedge$ size() $<$ $\backslash$old(entries.size()) $\wedge$ size() $\leq$ MAX))

Here, the tool issues a warning to the user, with the information that the original formula is valid, or vacuously satisfiable, on account of its inconsistent antecedent. Along the other branch, satisfiability of the negation of the consequent is queried, which is found to be satisfiable:

s *((result $\Rightarrow$ contains(e)) $\wedge$ entries = $\backslash$old(entries.add(e)*

At this point, the Conjunction Pattern can be applied to both the antecedent and the consequent to investigate the inconsistency of the former and to check whether the latter is vacuously satisfiable or contingent. In the prototype tool, automated analysis pauses here and the user prompted for continuation. It is likely that a user would be more interested in pursuing analysis of the inconsistent antecedent. Automated application of the Conjunction Pattern finds the inconsistent formulae to be *size() $<$ $\backslash$old(entries.size())* and *entries = $\backslash$old(entries.add(e))* (part of this step is the application of the Base Pattern to ascertain that these formulae are not internally inconsistent, i.e., that the method specifications of *size()* or *add()* are not individually inconsistent). Further automated applications of the Conjunction Pattern to the formulae in the specifications of *size()* and *add()* identifies the inconsistency to be the following:

entries.size() $>$ $\backslash$old(entries.size()) $\wedge$ size() $<$ $\backslash$old(entries.size())
$\wedge$ size() == entries.size())

The user should hopefully now see that the inconsistency is due to a typo in the evolved specification of *insert()*: the new postcondition from Section 2 should read

size() $>$ $\backslash$old(entries.size()) && size() $<=$ MAX

with a $>$ instead of $<$ in the first conjunct.

The soundness and completeness of pattern-driven analysis in practice of course depends on the soundness and completeness of the SAT-solver used in a given implementation. The Alloy Analyser is sound but not complete (the tool can run out of memory while undertaking certain queries). Practical considerations aside, pattern-driven analysis can be considered complete insofar as it terminates and finds the satisfiability values for any propositional formula and its subformulae. It will also issue a warning for any valid or unsatisfiable formula. Further, pattern-driven analysis can be considered sound insofar as the satisfiability values found for any propositional formula and its subformulae are correct, since these will come either from SAT or a sound inference from previous results [10].

Finally, the complexity of a pattern-driven analysis can be exponential. The worst case is the Conjunction Pattern, which may direct analysis through an exhaustive querying of each combination of conjunct in the root formula to determine a source of inconsistency. However, application of the Conjunction Pattern does not always take this branch and only does so when an inconsistency is present.

7. RELATED WORK

The concept of patterns to support an activity is of course not new. Design patterns [7] now provide handy templates for programmers across the world. Even in the area of formal specification, the idea of providing patterns to support specifiers has been considered before [6]. However, the patterns presented here are not patterns to be used by programmers or specifiers but the foundation of an automated analysis framework. They are templates for sound decision procedures whose implementation can be hidden from users.

Recent work on helping users of specification analysis tools avoid being misled by feedback includes [4] and [13], both dealing with JML. [4] presents an extension of an existing JML static checking tool which warns a user whenever a term in a precondition formula is undefined, thus avoiding cases where a method is spuriously reported to meet its specification simply because its precondition is unsatisfiable. The analysis patterns presented in this paper address similar concerns but are far more general in their application, since they check the satisfiability of arbitrary (propositional) formulae rather than focus on a particular cause of unsatisfiability such as undefinedness. [13] suggests including unsoundness and incompleteness disclaimers in feedback from automated analysis tools so that users are less likely to have misplaced confidence in feedback from unsound or incomplete analyses. Similar in spirit to the use of analysis patterns, this approach would be complementary to a patterns-based analysis toolset.

The notion of checking for vacuity is not new. For example, [2] addresses a form of vacuity checking in work on hardware verification. In the broader software verification setting, work also exists on catching vacuity in temporal model checking [14, 9].

Finally, the work in this paper is a greatly extended version of that published in [11]. This earlier work introduced the idea of analysis patterns but considered only cases of satisfiability and unsatisfiability. A contribution of the present paper is the definition of a lattice of six satisfiability values, which are used during pattern application. The feedback available from an application of the new analysis patterns therefore exceeds that provided by the previous versions.

8. CONCLUSION

This paper presented a set of automatable analysis patterns that constitute a framework for automated analysis of software specifications. The difficulties faced by developers and maintainers of a specification could be greatly reduced by an automated specification management environment. The analysis patterns and their prototype implementation are a step towards the provision of such an environment. In particular, the analysis patterns allow many hidden logical errors to be uncovered automatically, without need for expert direction and without offering misleading feedback. Due to a lack of space, two patterns for first order formulae (the *Universal-* and the *Existential Quantification Pattern* [10]), which primarily catch errors due to empty domains, have not been discussed here.

Further work will involve evaluating the approach with a full specification language such as JML, perhaps by extending an existing toolset. This would allow the scalability of the approach to be explored. Work in [10] on inference of satisfiability values from previous results could also be extended to provide further support for pattern-driven analysis. The more values that can be inferred, the fewer calls to a SAT-solver and the quicker an analysis. In addition, a non-trivial case study must be undertaken to investigate the limitations of the approach in application to industrial-scale specifications in practice.

Acknowledgements The authors wish to thank Michael Huth for his many comments on the work in this paper.

REFERENCES

[1] M Barnett, R Leino, and W Schulte. The Spec♯ Programming System: An Overview. *CASSIS*, 2004.

[2] Derek L. Beatty and Randal E. Bryant. Formally Verifying a Microprocessor Using a Simulation Methodology. *DAC*, 1994.

[3] L Burdy, Y Cheon, D Cok, M Ernst, J Kiniry, G Leavens, K Leino, and E Poll. An Overview of JML Tools and Applications. *STTT*, 2005.

[4] P Chalin. Early Detection of JML Specification Errors Using ESC/Java2. *SAVCBS*, 2006.

[5] K Dhara and G Leavens. Forcing Behavioral Subtyping Through Specifcation Inheritance. *ICSE*, 1996.

[6] M Dwyer, G Avrunin, and J Corbett. Property Specification Patterns for Finite-State Verification. *FMSP*, 1998.

[7] E Gamma, R Helm, R Johnson, and J Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. *Lecture Notes in Computer Science*, 707, 1993.

[8] J Gu, P Purdom, J Franco, and B Wah. Algorithms for the Satisfiability (SAT) Problem: a Survey. *Satisfiability Problem: Theory and Applications*, 1997.

[9] Arie Gurfinkel and Marsha Chechik. Extending Extended Vacuity. *FMCAD*, 2004.

[10] W Heaven. *Object-Oriented Specification: Analysable Patterns and Change Management*. PhD Thesis, Dept. of Computing, Imperial College London, 2007.

[11] W Heaven and A Russo. Enhancing the Alloy Analyzer with Patterns of Analysis. *WLPE*, 2005.

[12] D Jackson. *Software Abstractions*. The MIT Press, 2006.

[13] J Kiniry, A Morkan, and B Denby. Soundness and Completeness Warnings in ESC/Java. *SAVCBS*, 2006.

[14] Orna Kupferman and Moshe Y. Vardi. Vacuity Detection in Temporal Model Checking. *STTT*, 1999.

[15] G Leavens, A Baker, and C Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *Software Engineering Notes*, 31(3), 2006.

[16] LIFC. JML-Testing-Tools. http://lifc.univ-fcomte.fr/jmltt/.

[17] B Liskov and J Wing. A Behavioral Notion of Subtyping. *TOPLAS*, 1994.

[18] M Prasad, A Biere, and A Gupta. A Survey of Recent Advances in SAT-Based Formal Verification. *STTT*, 7(2), 2005.